A New Edit Distance for Fuzzy Hashing Applications

V. Gayoso Martínez, F. Hernández Álvarez, L. Hernández Encinas, and C. Sánchez Ávila

1Information Processing and Cryptography (TIC), Institute of Physical and Information Technologies (ITEFI) Spanish National Research Council (CSIC), Madrid, Spain
2Telecommunication Engineering School (ETSIT), Polytechnic University of Madrid (UPM), Madrid, Spain

Abstract—Similarity preserving hashing applications, also known as fuzzy hashing functions, help to analyse the content of digital devices by performing a resemblance comparison between different files. In practice, the similarity matching procedure is a two-step process, where first a signature associated to the files under comparison is generated, and then a comparison of the signatures themselves is performed. Even though ssdeep is the best-known application in this field, the edit distance algorithm that ssdeep uses for performing the signature comparison is not well-suited for certain scenarios. In this contribution we present a new edit distance algorithm that better reflects the similarity of two strings, and that can be used by fuzzy hashing applications in order to improve their results.

Keywords: Edit distance, fuzzy hashing, similarity preserving hashing

1. Introduction

Similarity Preserving Hashing (SPH) functions, also known as fuzzy hashing algorithms, try to detect the resemblance between two files [1]. There are basically four types of SPH functions [2]: Block-Based Hashing (BBH) functions, Context-Triggered Piecewise Hashing (CTPH) functions, Statistically-Improbable Features (SIF) functions, and Block-Based Rebuilding (BBR) functions. In any fuzzy hashing application, files are processed and, as a result of the analysis performed, a code linked to the content of the file is generated, so files can be later compared based on their codes. In this context, the file’s code is indistinctly referred to as its digest, hash or signature.

In CTPH functions, the length and content of the signature is determined by the existence of certain special points, called trigger points or distinguished points, within the data object. A point is considered to be a trigger point if it matches a certain property, defined in a way so that the number of expected trigger points falls within a previously specified range. Once a number of trigger points large enough is detected, CTPH applications generate the signature associated to the file by processing the data portions located between consecutive trigger points.

Since its first release, ssdeep [3] has been one of the best known fuzzy hashing applications. When comparing files, ssdeep generates a matching score after analysing the similarity of the signatures. In order to do that, ssdeep implements an edit distance algorithm based on the Damerau-Levenshtein distance between two strings [4], [5]. That edit distance function compares the two strings and counts the minimum number of operations needed to transform one into the other, where the allowed operations are insertions, deletions, and substitutions of a single character, and transpositions of two adjacent characters [6], [7].

Even though the success of ssdeep is quite remarkable, its edit distance implementation has important limitations that prevent ssdeep from generating a score that reflects the percentage of the bigger file that is also present in the smaller file, which is the definition of similarity better adapted for some real-world scenarios. With the goal to improve the quality of fuzzy hashing applications, in this contribution we present a new edit distance algorithm that can be used as a replacement of ssdeep’s edit distance or in new implementations.

Even though the success of ssdeep is quite remarkable, its edit distance implementation has important limitations that prevent ssdeep from generating a score that reflects the percentage of the bigger file that is also present in the smaller file, which is the definition of similarity better adapted for some real-world scenarios. With the goal to improve the quality of fuzzy hashing applications, in this contribution we present a new edit distance algorithm that can be used as a replacement of ssdeep’s edit distance or in new implementations.

The rest of this paper is organized as follows: Section 2 reviews ssdeep’s edit distance. In Section 3, we provide a complete description of our proposed algorithm. Section 4 includes a comparison of both algorithms when working with some special signatures. Finally, Section 5 summarizes our conclusions about this topic.

2. Edit distance in ssdeep

In 2006, Jesse Kornblum released ssdeep [8], one of the first programs for computing context triggered piecewise hashes, and that soon became very popular. Since that initial release, new versions and updates have not ceased to appear, and the project is still active (at the time of preparing this contribution, the latest version is 2.12, which was released in October 2014 [3]). The core of ssdeep is derived from rsync[9] and spamsum[10], both of them tools developed by Andrew Tridgell.

As mentioned in the previous section, the similarity measurement that ssdeep uses is an edit distance algorithm based on the Damerau-Levenshtein distance [4], [5], [6], [7]. In the original Damerau-Levenshtein algorithm, all the operation costs are initially 1, though the substitutions and transpositions decrease their weight to 0 when certain conditions are met [11]. In comparison, ssdeep defines the weight of insertions and deletions as 1, the weight of substitutions as 3, and the weight of transpositions as 5.
As an example, using ssdeep’s algorithm the distance between the strings “emerald” and “overall” is 6, as it can be checked with the following steps and the computations of Table 1.

```
emerald  del m erald  del e rald  ins y erald  ins rerald
overall  del o veral  ins o veral
```

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>e</th>
<th>m</th>
<th>r</th>
<th>a</th>
<th>l</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>v</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>r</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>l</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1: ssdeep edit distance example.

A consequence of assigning the weights 3 and 5 to the substitution and transposition operations is that, in practice, the edit distance computed by ssdeep only takes into consideration insertions and deletions. In this way, a substitution has a cost of 2 (a deletion plus an insertion) instead of 3, and a transposition has also a weight of 2 (again an insertion and a deletion) instead of 5.

One of the limitations derived from this design is that, given a string, a rotated version of the initial string is credited with many insertion and deletion operations, when in its nature it is basically the same string (i.e. the content is the same, although the order of the substrings is different). Consider for example the strings “1234abcd” and “abcd1234”.

As the signature comparison algorithm implemented by ssdeep is not available in a descriptive way, Algorithm 1 shows our interpretation (made upon inspection of the source code of ssdeep [3]) of that functionality, where A and B are one-dimensional arrays containing, respectively, the m characters of string1 and the n characters of string2, and where D is a \((m + 1) \times (n + 1)\) matrix used in the computations with all its positions initially set to 0. During the set-up phase, the first row (respectively, the first column) of D is initialized with the number corresponding to the column (respectively, the row) of the position being processed. The rest of the positions are processed based on the content of the nearby elements and the characters being compared. Once the comparison procedure is finished, the algorithm generates a similarity score in the range 0-100.

The meaning of the functions included in the algorithm is the following:

- `length(string)`: calculates the number of characters of the string.
- `longestCommonSubString(string1,string2)`: provides the longest common substring of two strings.
- `min(param1,param2,param3)`: identifies the minimum value given by the numbers or expressions passed to the function as parameters.
- `floor(value)`: returns the bigger integer whose value is equal to or lower than value.

3. Our proposed edit distance

Our edit distance algorithm compares two signature strings, string1 and string2, and produces a similarity score in the range 0-100. Algorithm 2 describes all the steps that must be performed in order to evaluate the similarity of the strings string1 and string2, where the first step consists in identifying as string1 the shortest string and as string2 the longest string, swapping the strings if necessary. During the procedure, the algorithm manipulates modified versions of the input strings, using their longest common substring for deciding which modification to perform next and increasing a counter with the differences found so far. The procedure is repeated until there are no more common substrings for the modified versions of the input elements. In the final step, the algorithm compares the resulting strings character by character in order to add to the counter the number of difference elements found for the same positions. It is important to point out that, unlike ssdeep, our algorithm does not impose a minimum length for the longest common substring, which allows to compare a wider range of strings.

The meaning of the functions included in Algorithm 2 and not presented in the previous section is the following:

- `longestCommonSubStringNoHyphen(string1,string2)`: returns the longest common substring which does not contain the hyphen (-) character.
- `hyphenString(size)`: creates a new string of length size containing only the hyphen character.
- `indexOf(string,substring)`: returns the position where the first character of substring is located inside string.
- `replace(string,index,size,string)`: replaces in the element string the existing substring of size characters starting at index with the characters of substring.
- `abs(number)`: provides the absolute value of the input number.
- `charAt(string,index)`: returns the character located at position index in the element string.

In order to illustrate the comparison process performed by Algorithm 2, Table 2 provides an example using two ad-hoc strings, denoted as string1 and string2. In the first row of the table, we have included the two initial strings (renamed as string1temp and string2temp), the template for the modified version of string2 (called string2mod), and the score, which initially equals 0. Starting with the step 1, the element substring identifies the longest common substring of string1temp and string2temp, which are then updated to show the removal of that substring. Then, we have inserted the common
Algorithm 1 ssdeep edit distance algorithm.

1: if (length(longestCommonSubstring(string1,string2)) < 7) then
2:    return 0
3: end if
4: \( \lambda_{del} \leftarrow 1 \)
5: \( \lambda_{ins} \leftarrow 1 \)
6: \( \lambda_{sub} \leftarrow 3 \)
7: i \leftarrow 0
8: for all \( i \leq m \) do
9:    D[i,0] \leftarrow i
10: end for
11: j \leftarrow 0
12: for all \( j \leq n \) do
13:    D[0,j] \leftarrow j
14: end for
15: i \leftarrow 1
16: j \leftarrow 1
17: for all \( i \leq m \) do
18:    for all \( j \leq n \) do
19:        if (A[i] = B[j]) then
20:            \( \lambda_{sub} \leftarrow 0 \)
21:        else
22:            \( \lambda_{sub} \leftarrow 3 \)
23:        end if
24:        D[i, j] = min(D[i-1,j] + \( \lambda_{ins} \), D[i,j-1] + \( \lambda_{del} \), D[i-1,j-1] + \( \lambda_{sub} \))
25:    end for
26: end for
27: score \leftarrow D[m,n]
28: score \leftarrow floor(\frac{score \cdot 100}{length(string1)+length(string2)})
29: if (score > 100) then
30:    score \leftarrow 0
31: else
32:    score \leftarrow 100 - score
33: end if
34: return score

longest substring obtained in that step into \( string2mod \), so the position of that substring in \( string2mod \) is the same that it occupies in \( string1 \).

As described in Algorithm 2, we only increase the score if the difference between the initial and final positions of the substring in \( string2mod \) is greater than the length difference of \( string1 \) and \( string2 \). With this rule we avoid to penalize the change of positions derived from the different length of the strings under comparison (e.g. this difference could have been produced by the insertion of some characters at the beginning of the string, which would displace the rest of the characters that compose the original string a given number of positions).

The score is increased in one unit if the longest common substring has more than one character, which means that common substrings of different sizes would receive the same penalty (i.e., a penalty of 1.0, but only if they are separated a number of positions bigger than the difference of the string lengths). In this sense, what we penalize is the movement of the string, not its size.

Besides, when the longest common substring has exactly one character, the quantity to be added to the score is 0.5. The reason for doing this is not to penalize in excess the displacement of a unique character. If we do not impose this rule, the displacement of a single character would receive a score of 1.0, which would be the same penalty produced by the substitution of a character by a completely different character. A topic open for future study is the modification of this value in order to obtain better results.

After the rearrangement phase, a pair by pair comparison of the characters elements is performed in the last step of the procedure. As there are eight different characters
Algorithm 2 Our proposed edit distance algorithm.

```
1: if (length(string1) > length(string2)) then
2:   string1 ↔ string2
3: end if
4: string1temp ← string1
5: string2temp ← string2
6: common ← longestCommonSubStringNoHyphen(string1temp, string2temp)
7: string2mod ← hyphenString(length(string2))
8: diff ← 0
9: while (length(common) > 0) do
10:   pos1 ← indexOf(string1temp, common)
11:   pos2 ← indexOf(string2temp, common)
12:   string2mod ← replace(string2mod, pos1, length(common), common)
13:   if (abs(pos1-pos2) > abs(length(string1)-length(string2))) then
14:     diff ← diff + 1
15:   else
16:     diff ← diff + 0.5
17:   end if
18: end while
19: string1temp ← replace(string1temp, pos1, length(common), hyphenString(length(common)))
20: string2temp ← replace(string2temp, pos2, length(common), hyphenString(length(common)))
21: common ← longestCommonSubStringNoHyphen(string1temp, string2temp)
22: for all i such that 0 ≤ i ≤ length(string2temp) do
23:   char ← charAt(string2temp, i)
24:   if char ≠ "-" then
25:     pos2 ← indexOf(string2mod, "-")
26:     string2mod ← replace(string2mod, pos2, 1, char)
27:   end if
28: end for
29: for all i such that 0 ≤ i ≤ length(string2temp) do
30:   if ((i ≥ length(string1)) or (charAt(string1, i) ≠ charAt(string2mod, i))) then
31:     diff ← diff + 1
32: end if
33: end for
34: return floor(100 - \frac{diff \cdot 100}{length(string2)})
```

in string1 and string2mod, the score is increased in eight units from 4.5 up to 12.5. In order to facilitate the identification of the dissimilar characters, Table 2 displays in bold font the dissimilar elements of the two strings.

Taking into account that the length of the longest string (string2) is 27, the comparison between string1 and string2 provides the following output:

\[
\text{Result} = 100 - \left\lfloor \frac{12.5 \cdot 100}{27} \right\rfloor = 100 - 46 = 54.
\]

A score of 54 implies that 54% of the longest string, string2, is also contained in the shorter string, string1.

4. Special signatures

When designing this test, our goal was to check the behaviour of ssdeep’s algorithm and our proposed algorithm when using some special strings, whose pattern could appear in certain real-world scenarios (for example, when obtaining the signature of files containing lists of elements such as names, file paths, etc.).

Even though we are aware that the tests included below represent extreme cases with ad-hoc strings, we believe it is worthwhile to test both algorithms in this scenario, as it represents different degrees of content rotation and modification.
The strings included in this test are the following ones:

\[ \text{S01: } \text{BCDEFGHIJKLMNOPQRSTUVWXYZ} \]
\[ \text{S02: } \text{LMNOPQRSTUVWXYZ} \]
\[ \text{S03: } \text{abcdEFGHIJKLMNOPQRSTUVWXYZ} \]
\[ \text{S04: } \text{12345678901234567890123456} \]
\[ \text{S05: } \text{BADCFEHGJILKNMPORQTSVUXWYZ} \]
\[ \text{S06: } \text{CDABGHEFKJILMNPQRSTUVWXYZ} \]
\[ \text{S07: } \text{EFGHABCDMNOPIJKLMNPQRSTUVWXYZ} \]
\[ \text{S08: } \text{IJKLMNOPQRSTUVWXYZ} \]
\[ \text{S09: } \text{QRSTUWXYZ} \]
\[ \text{S10: } \text{ghijklmnopqrstuvwxyz} \]

The first string, S01, can be considered the base element of the set. The second string replaces the second half of S01 with its own first half. String S03 swaps the two blocks that form S01. In addition to the previous change, string S04 replaces the first half of the string with digits. Strings S05 to S10 take as basis the first string and perform transpositions of blocks whose size is 1, 2, 4, 8, 16, and 32 characters, respectively.

The results generated when comparing these special signatures are included in Tables 3, 4, and 5. Table 3 shows the results obtained when using ssdeep with signature files whose content replies the strings of the tests. As the limitation imposed by ssdeep regarding the minimum length for the common substrings produces as a result that several comparisons are not effectively performed (ssdeep directly assigns a score of 0 in those cases), we have implemented the logic of ssdeep’s algorithm in Java Standard Edition [12] and have removed that limitation in our code. Thus, Table 4 shows the results that ssdeep would provide if it did not apply the aforementioned minimum length requirement. Finally, Table 5 displays the results obtained with our proposed algorithm once implemented as another Java application.

As it can be observed, our algorithm is able to provide meaningful results in all the comparisons, which is not the case in ssdeep. For example, the comparison between S01 and S07, which renders a score of 0 in ssdeep, is evaluated as having a similarity degree of 77% by our algorithm. Following that example, the modified version of ssdeep without the minimum length requirement generates a score of 55 which, even representing a better result, it still fails to properly reflect the fact that S01 and S07 share far more than half of their content.

When inspecting the tables, it can be stated that the results provided by our algorithm are more realistic according to the similarity definition given in the Introduction. For instance, when comparing S01 to S03 and S04, it is clear that S03 is almost the same string as S01, whilst S04 only shares half of its string. However, ssdeep is not able to detect that difference and assigns a value of 50% in both cases. In comparison, our algorithm computes the similarity degree as 97% and 49%, respectively.

Even though the modified version of ssdeep provides higher results than our algorithm in some instances (e.g., when comparing S02 and S05 or S04 and S06), those differences are small and do not imply a representative difference. However, when our algorithm provides higher results the difference in some instances is quite important (e.g., when processing S08 and S09). In fact, the average difference in the scores of the test when comparing different strings is 18.96 in favour of our method. We are aware that, in general, a higher value should not imply a better result; however, when comparing the test strings, which clearly share an important part of their contents, a higher result implies a better similarity detection capability.

Table 2: String rearrangement example.
Table 3: Test results for special cases with ssdeep.

<table>
<thead>
<tr>
<th></th>
<th>S01</th>
<th>S02</th>
<th>S03</th>
<th>S04</th>
<th>S05</th>
<th>S06</th>
<th>S07</th>
<th>S08</th>
<th>S09</th>
<th>S10</th>
</tr>
</thead>
<tbody>
<tr>
<td>S01</td>
<td>100</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>55</td>
<td>63</td>
<td>63</td>
</tr>
<tr>
<td>S02</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>47</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>S03</td>
<td>50</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>36</td>
<td>44</td>
<td>90</td>
</tr>
<tr>
<td>S04</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>32</td>
<td>50</td>
</tr>
<tr>
<td>S05</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S06</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S08</td>
<td>63</td>
<td>47</td>
<td>36</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>S09</td>
<td>63</td>
<td>50</td>
<td>44</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>S10</td>
<td>63</td>
<td>50</td>
<td>90</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>32</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 4: Test results for special cases with modified version of ssdeep.

<table>
<thead>
<tr>
<th></th>
<th>S01</th>
<th>S02</th>
<th>S03</th>
<th>S04</th>
<th>S05</th>
<th>S06</th>
<th>S07</th>
<th>S08</th>
<th>S09</th>
<th>S10</th>
</tr>
</thead>
<tbody>
<tr>
<td>S01</td>
<td>100</td>
<td>50</td>
<td>97</td>
<td>49</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>77</td>
<td>89</td>
<td>93</td>
</tr>
<tr>
<td>S02</td>
<td>50</td>
<td>100</td>
<td>49</td>
<td>49</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>37</td>
<td>43</td>
<td>47</td>
</tr>
<tr>
<td>S03</td>
<td>97</td>
<td>49</td>
<td>100</td>
<td>50</td>
<td>50</td>
<td>74</td>
<td>74</td>
<td>74</td>
<td>85</td>
<td>91</td>
</tr>
<tr>
<td>S04</td>
<td>49</td>
<td>49</td>
<td>50</td>
<td>100</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>37</td>
<td>43</td>
<td>47</td>
</tr>
<tr>
<td>S05</td>
<td>50</td>
<td>25</td>
<td>50</td>
<td>25</td>
<td>100</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>S06</td>
<td>50</td>
<td>25</td>
<td>50</td>
<td>25</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>S07</td>
<td>77</td>
<td>37</td>
<td>74</td>
<td>37</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>77</td>
<td>79</td>
<td>75</td>
</tr>
<tr>
<td>S08</td>
<td>89</td>
<td>43</td>
<td>85</td>
<td>43</td>
<td>50</td>
<td>50</td>
<td>77</td>
<td>100</td>
<td>87</td>
<td>87</td>
</tr>
<tr>
<td>S09</td>
<td>93</td>
<td>47</td>
<td>91</td>
<td>47</td>
<td>50</td>
<td>50</td>
<td>79</td>
<td>87</td>
<td>100</td>
<td>93</td>
</tr>
<tr>
<td>S10</td>
<td>97</td>
<td>49</td>
<td>97</td>
<td>49</td>
<td>50</td>
<td>50</td>
<td>75</td>
<td>87</td>
<td>93</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5: Tests results for special cases with our algorithm.

5. Conclusions

In this contribution we have presented a new edit distance algorithm that can be used in fuzzy hashing applications. Our algorithm provides better results than ssdeep’s algorithm according to a definition of similarity useful in computer forensics when comparing two files, and that interprets similarity as the percentage of a file that is also present in another file. We have implemented both our algorithm and a modified version of ssdeep in Java, and have used those two applications together with version 2.12 of ssdeep in order to test some strings that could represent the signature of files including a list of elements.

The tests performed with the three applications allow us to state that our algorithm provides results better adapted to the aforementioned definition of similarity, so it can be considered as an alternative for the edit distance currently implemented in ssdeep and other fuzzy hashing applications.
Acknowledgment

This work has been partially supported by Comunidad de Madrid (Spain) under the project S2013/ICE-3095-CM (CIBERDINE) and by Ministerio de Economía y Competitividad (Spain) under the grant TIN2014-55325-C2-1-R (ProCriCiS).

References


